An Architectural Support for Self-Adaptive Software for Treating Faults

Rogério de Lemos
Computing Laboratory
University of Kent at Canterbury, UK
r.delemos@ukc.ac.uk

José Luiz Fiadeiro
ATX Software S.A, and LabMOL–University of Lisbon
Alameda António Sérgio 7 – 1 C
2795-023 Linda-a-Velha, Portugal
jose@fiadeiro.org

ABSTRACT
Capitalising on previous work on building systems from existing components that allow neither code inspection nor change, an architectural approach for treating faults through self-adaptation is outlined that, instead of providing mechanisms and techniques at the component level, relies on the interactions between components for obtaining flexible software structures that are nevertheless robust to the occurrence of undesirable events.

Categories and Subject Descriptors
D.2.11 [**Software Architectures**]: Languages, patterns.

D.4.5 [**Reliability**]: Fault tolerance.

General Terms
Design, Reliability, Languages.

Keywords
Software architectures, dependability, fault tolerance, fault treatment, dynamic adaptation.

1. INTRODUCTION
As software systems become more complex, ridding their execution from the occurrence of faults can only be a false promise or assumption. Hence, when building systems, provisions have to be made to include mechanisms and techniques that will enable them to operate in spite of the presence of faults.

In this paper, we suggest the adoption of an architectural approach for treating faults in complex software systems that is centred on interactions between components, The focus is on the isolation of faulty components and reconfiguration of system structure, assuming that mechanisms for fault identification and localisation are already available.

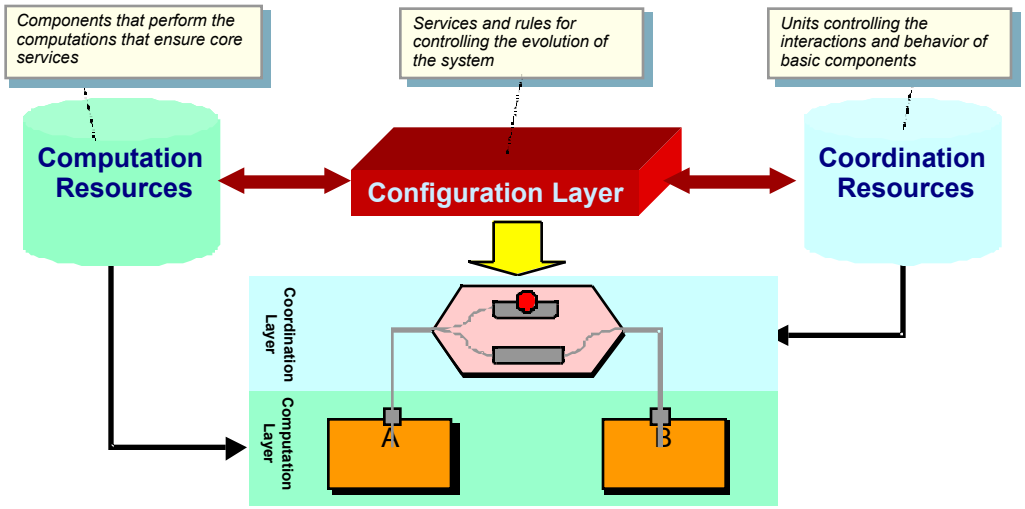
The emphasis on interactions is justified by the fact that, in order to achieve the levels of flexibility and responsiveness required for self-adaptation, fault treatment needs to be directed not to the core components of the system but to the interconnections through which they interact. "Repairing" components is normally either impossible because they are "black-boxes", or difficult to control because it can have side effects, be error-prone and time-consuming. "Replacing" components with equivalent ones may also be impossible because redundancies may not be available for all kinds of required services. Acting on the interactions, making them "light-weight" architectural connectors that can be easily plugged in and out of system configurations, provides more control on the adaptation process because it makes it compositional over the architecture of the system.

Furthermore, by adopting technologies based on the separation between computation, coordination, and configuration we can provide the means for such systems to be reconfigurable in run-time, without interruption of service. This is general enough to address operations like isolating or modifying the behaviour of a faulty component, and expressive enough to support a reactive approach to reconfiguration that can handle self-adaptation, what sometimes is called "self-healing". Our aim is to define a coordination-based configuration language in which we can program ways in which the system can react to events that arise from the occurrence of faults by executing "configuration transactions". By this we mean collections of elementary reconfiguration operations that need to be collectively executed in an atomic way [9], but not in any prescribed order: this order should emerge from the nature of the operations to be executed and the context in which they need to be executed. This is important in order to guarantee that the system can react to "unexpected" situations but through "predictable" means.

In some applications, the resources from which solutions can be pulled are normally limited or constrained. Hence, reconfiguration is not a mere replacement of components by "equivalent" ones, but requires a revision of the connectors that related the faulty component to the rest of the system so that it "adapts" the replacement to the expectations that other components of the system have on the services that it will provide. In the architectural approach that we propose, a component may not be able to support a given set of service requirements but, suitably adapted, may be able to contribute to a downgraded, but acceptable, level of service.

The rest of the paper is structured as follows. Section 2 gives a brief overview of fault tolerance. Section 3 describes the proposed architectural solution for treating faults. Section 4 briefly describes some related work. Finally, the conclusions are presented in section 4.

WOSS '02, Nov 18-19, 2002, Charleston, SC, USA.



2. FAULT TOLERANCE

Dependability is a vital property of any system justifying the reliance that can be placed on the service it delivers [8]. *Fault tolerance* is a means of achieving dependability, working under the assumption that a system contains faults (e.g. ones made by humans while developing or using systems, or caused by aging hardware), and aiming at providing the specified services in spite of their presence. Fault tolerance together with fault avoidance, removal and forecasting are known as dependability means.

The causal relationship between the dependability impairments, that is, faults, errors and failures, is essential to characterise the major activities associated with fault tolerance. A *fault* is the adjudged or hypothesized cause of an error. An *error* is the part of the system state that is liable to lead to the subsequent failure. A *failure* occurs when a system service deviates from the behaviour expected by the user.

Fault tolerance is carried by *error processing*, aiming at removing errors from the system state before failures happen, and *fault treatment*, aiming at preventing faults from being once again activated. Error processing typically consists of three steps: error detection, error diagnosis and error recovery. Fault treatment consists of *fault diagnosis*, which determines the causes of the error, in terms of location and nature, and *fault removal*, which isolates the faulty components to avoid the reactivation of faults. The removal of faults may consist in reconfiguring the system by modifying its structure in order for the system to continue to deliver an acceptable service.

3. PROPOSED ARCHITECTURE

The approach being proposed in this paper is based on a set of modelling primitives that have been recently proposed for an architecture-based approach to evolution [1,2,5]. It is based on the enforcement of a strict separation of three layers: *computation* that manages the computations that are performed locally within components, *coordination* that is responsible for enforcing the interactions between components that are required for global system properties to emerge, and *configuration* that determines when and how the components and connectors should be linked. This strict separation allows for each level to be managed independently of the other, which leads to the required degree of dynamic reconfigurability. Hence, this approach will make a significant advance to the state of the art in fault treatment because it will target specifically the interconnections rather than the components. However, the architectural-based approaches to evolution mentioned above need, nevertheless, to be extended in order to incorporate mechanisms for supporting dynamic reconfiguration, including the means for modelling and evaluating configuration transactions in a stable way.

3.1 Separating Computation, Coordination and Configuration

The need to operate, in "real-time", with "surgical" precision for limiting the impact of the treatment, in contexts of increasing interdependency, requires a clear separation of concerns to be enforced in the way we model and manage such systems. The proposed three-layered architecture that separates what we consider to be the key concerns involved in this problem is shown in figure 1.

The separation between computation and coordination is enforced by modelling explicitly the interactions that exist in the system as first-class entities – architectural connectors that we call coordination contracts. These connectors coordinate the way the components that reside in the computation layer interact. The latter correspond to "core" entities of the domain that provide basic services that, usually, cannot be "repaired" because they are performed by "black-boxes". By externalising all interactions as connectors, it becomes possible to circumscribe treatment of faults occurring at the level of a component to the connectors through which it interacts with the rest of the system [5]. Basically, because it is often impossible to find a component that performs "equivalent" services, we see fault treatment as consisting of searching, within the available resources, components that offer alternative services, even if in a downgraded mode, and establishing the connectors that can adapt them to the expectations of the components with which they are required to interact.

This model supports the means for fault treatment to be performed through dynamic reconfiguration, in run-time, without interruption of service. For this process of

reconfiguration to be able to be programmed, leading to self-adaptive and self-healing features, we propose a third architectural layer consisting of entities that can react to events and act on the configuration, which are treated, again, as first-class citizens.

3.2 Fault Treatment

A key characteristic of several software systems is the need to continuously provide acceptable level of service, even in the presence of faults. The ability of a system to deliver its required service relies on its capability of isolating faults and being reconfigured in order to avoid faults from being activated again, which might lead to the eventual failure of the system as a whole. The successful reconfiguration of a system depends on the availability of redundancies, the ability to modify system structure, and the definition of acceptable (but less desirable) levels of service [8]. The reconfiguration of systems can be achieved but is not limited to the provision of mobility through resource location, design diversity through implementation, and dynamic architectures through interconnection [13]. The approach being proposed relies on the dynamic reconfiguration of the architecture, in which the diversity in the services provided by the system components might provide the appropriate redundancies for certain classes of failure.

A potential solution would be to perform system reconfiguration through a sequence of atomic transactions until a stable state of the system configuration is achieved. However, in between fault localisation and the end of system reconfiguration, several decisions have to be taken depending on the resources available in the system and the services to be provided by the system.

4. RELATED WORK

The proposed architectural approach tries to address the challenging problem of reconfiguring complex software systems in the presence of faults. Although some work has been done in adaptive fault tolerance [6,7], most of this work did not consider fault treatment in the context of complex large-scale systems, where redundancies for the provision of fault tolerance may not be available at the system support level. In the proposed approach, the provisions for tolerating faults should rely on redundancies in the services provided by the system components. Very few studies have considered the description of software architectures with respect to their non-functional properties, and in particular dependability properties [10,11,12]. Moreover, these do not cover the issues specifically for representing, at the architectural level, system reconfiguration for the purpose of fault treatment.

In the context of software architectures, some efforts have been made for providing mechanisms for monitoring and controlling the actual execution of a system through its architectural model, thus allowing self-healing/self-repair of the system at higher levels of abstraction. One of these initiatives relies on extending existing architectural styles by incorporating constraints that capture the desired behaviour of the system [3].

5. CONCLUSIONS

While most of the work in fault-treatment has been directed to software components, we suggested in this paper a focus on component interconnections as a means of achieving hiher levels of architectural flexibility for supporting run-time adaptability.

Although the suggested approach is based on modelling techniques that we have developed, formalised and applied in the design of complex software systems, their application to fault treatment poses a series of new challenges. In the context of fault treatment: how to identify available redundancies, how to instantiate system and component level reconfiguration policies into strategies, and how to realize a strategy without disrupting the system service, just to name a few. While, in terms of software architectures: how to represent reconfiguration policies and strategies at the component and system level, how to represent at the interface of the components the services they should be able to provide, including their non-functional properties, and how to incorporate into the architectural models mechanisms for evaluating and validating architectural configuration during run-time. These challenges are at the core of a research programme that we intend to pursue in the near future, namely through the application of the proposed techniques to concrete case studies.

REFERENCES

[1] L. Andrade and J. Fiadeiro. “Coordination Technologies for Managing Information System Evolution”. *Proceedings of CAiSE'01*. K. Dittrich, A. Geppert and M. Norrie Eds. LNCS 2068. Springer-Verlag. 2001.pp. 374-387.

[2] L. Andrade, and J. Fiadeiro. “Coordination: The Evolutionary Dimension”. *Proceedings TOOLS Europe 2001*. Ed. W. Pree. IEEE Computer Society Press. 2001.pp. 136-147.

[3] S.-W. Cheng, D. Garlan, B. Schmerl., J. Sousa, B. Spitznagel, and P. Steenkiste. “Using Architectural Style as the Basis for Self-repair”. *The Working IEEE/IFIP Conference on Software Architecture 2002*. Montreal, Canada. August 2002. (to appear)

[4] R. de Lemos. “A Co-operative Object-Oriented Architecture for Adaptive Systems”. *Proceedings of the 7th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS’00)*. Edinburgh, Scotland. April 2000. pp. 120-124.

[5] R. de Lemos. “Describing Evolving Dependable Systems using Co-operative Software Architectures”. *Proceedings of the IEEE International Conference on Software Maintenance (ICSM’02)*. Florence, Italy. November 2001. pp. 320-329.

[6] J. Goldberg, L. Gong, I. Greenberg, R. Clark, E. D. Jensen, K. Kim and D. Wells. *Adaptive Fault-Resistant Systems*. SRI Technical Report. 1994.

[7] M. A. Hiltunen and R. D. Schlichting. “Adaptive Distributed and Fault-Tolerant Systems”. *International Journal of Computer Systems and Engineering 11(5)*. 1995. pp. 125-133.

[8] J.-C. Laprie. “Dependability: Basic Concepts and Terminology”. *Special Issue of the Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS- 25)*. IEEE Computer Society Press, 1995.pp. 42-54.

[9] P. Oriezy, M. M. Gorlick, R. N. Taylor, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. “An Architecture-Based Approach to Self-Adaptive Software”. *IEEE Intelligent Systems 14(3)*. May/June 1999. pp. 54-62.

[10] T. Saridakis and V. Issarny. “Developing Dependable Software Systems using Software Architectures”. *Software Architecture*. Kluwer Academic Publisher. 1999.

[11] D. Sotirovski. “Towards Fault-Tolerant Software Architectures”. *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*. Amsterdam, The Netherlands. pp. 7-13. August 2001.

[12] V. Stavridou and R.A. Riemenschneider. “Provably Dependable Software Architectures”. *Proceedings of the Third ACM SIGPLAN International Software Architecture Workshop*. pp. 133-136. 1998.

[13] K. Sullivan, J. C. Knight, X. Du and S. Geist. “Information Survivability Control Systems”. *Proceedings of the 21st International Conference on Software Engineering (ICSE’99)*. Los Angels, CA. May 1999.pp. 184-192.